

# ECE 741 / 841

12 September 2002

## Higher Order Logics

So far, we have been dealing with propositional logic and 1st order logic. We can extend 1st order logic by allowing quantification over predicates and functions. This will give us a 2nd order logic.

Example:  $\forall f \forall x \forall g f(x) = g(x) \rightarrow f = g$

If we introduced functions of functions, quantification over function of functions and quantification over types, we could have logics of 3rd order, 4th order, etc.

We will introduce higher order notions and notations as needed.

## Lambda Notation

A lambda expression is a way to define a function in which the function is an operator. Example operators are derivative ( $\frac{\partial}{\partial t}$ ) or Laplace transform ( $\mathcal{L}$ ).

A function can be defined in the traditional way:

$$f(x) = x^2$$

or by a lambda expression:

$$\lambda x : x^2$$

In a lambda expression, there is not an equality. When a function is passed as an argument to another function, a lambda expression is very convenient.

## Lambda Notation, cont.

$$(\lambda x : x^2)3 = 3^2$$

A function  $g$  takes a functions (unary) and two objects and applies the function to the two objects, add them, and then applies the function to the result:

$$g(f_1, x, y) = f_1(f_1\ x + f_1\ y)$$

We can use our function above:

$$g(\lambda x : x^2, x, y)$$

We could not use:

$$g(f(x) = x^2, x, y)$$

## **Lambda Notation, cont.**

Note that we have used a combination of function definition. The function  $g$  is defined in the standard way.  
The arguments to  $g$  are lambda expressions.

## Partial Application

Lets define a function with two arguments:

$$\lambda x\ y : y^x$$

We can apply this function to two objects:

$$((\lambda x\ y : y^x)3)4 = 4^3$$

It is also possible to apply a lambda function partially:

$$((\lambda x\ y : y^x)2 = \lambda y : y^2$$

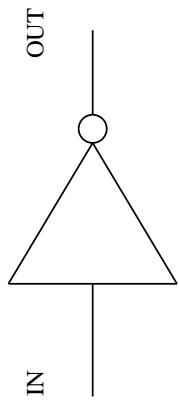
The result is a function of one argument.

## Modeling Gates

We can use our formal language to model digital circuits.  
We are going to model digital circuits at the discrete level of abstraction. We are not going to model physical characteristics.

- Noise
- Temperature
- Capacitance
- Inductance, etc.

## Modeling Bits in Logic (Selecting the Type)



in, out: {true, false}

in, out: {0,1}

in, out: {L,H}

in, out: {L,H,X}

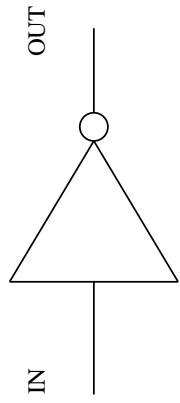
in, out: [nat -> bool]

in, out: [nat -> {0,1}]

in, out: [real -> bool]

etc.

## Specification Style



- Assertion

$\text{INV}(\text{in}, \text{out}) = (\text{out} = \neg \text{in})$

- Functional

$\text{INV}(\text{in}) = \neg \text{in}$

- Timed Assertion

$\text{INV}(\text{in}, \text{out}) = \forall t: (\text{out}(t+1) = \neg \text{in}(t))$

## The Meaning of an Assertional Specification

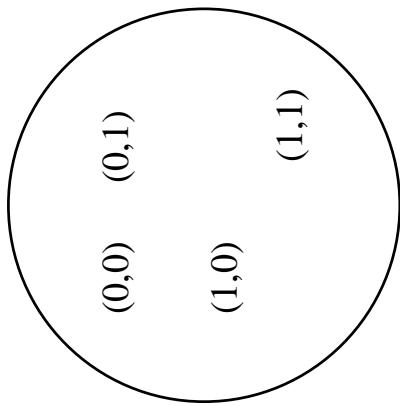
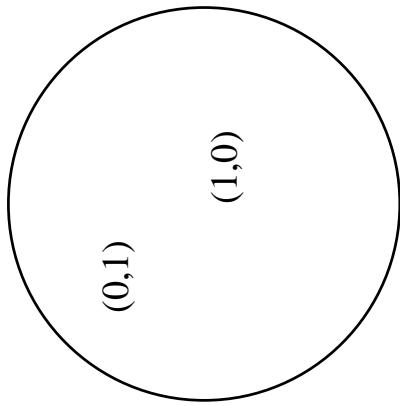
Assuming  $\text{in}, \text{out} \in \{0, 1\}$

in	out	$\text{INV}(\text{in}, \text{out})$
0	0	false
0	1	true
1	0	true
1	1	false

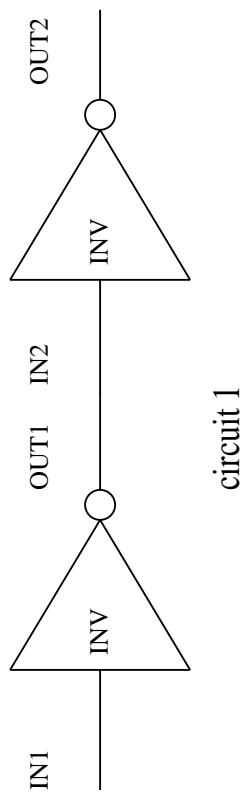
## The Meaning of an Assertion Specification

The predicate  $\text{INV}(\text{in}, \text{out})$  defines a relation between  $\text{in}$  and  $\text{out}$  which can be represented by a set of pairs.

$\text{INV}(\text{in}, \text{out})$



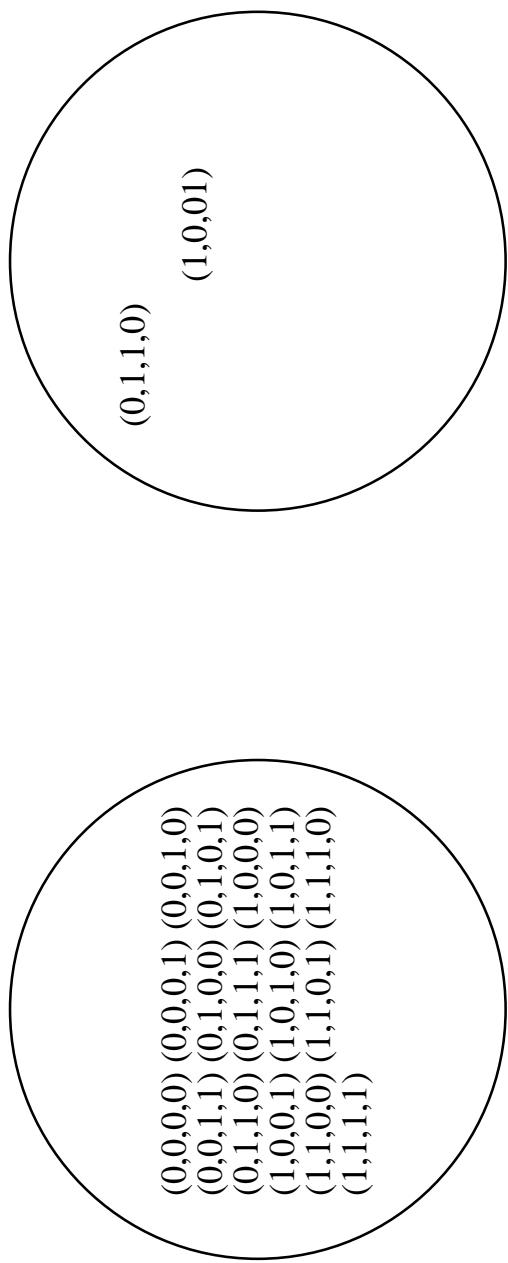
## Connecting Gates



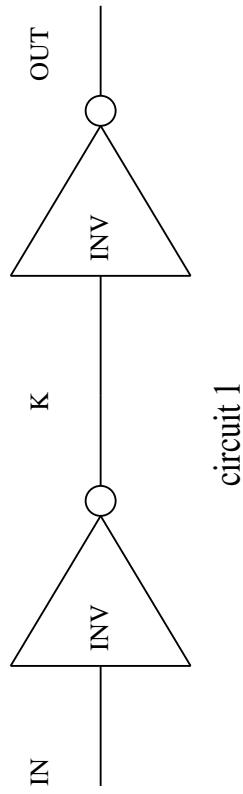
How would we model or represent the connection of these two gates?

## Connecting Gates

$\text{circuit1}(\text{in1}, \text{out1}, \text{in2}, \text{out2}) = \text{INV}(\text{in1}, \text{out1}) \wedge$   
 $\text{INV}(\text{in2}, \text{out2}) \wedge \text{out1} = \text{in2}$

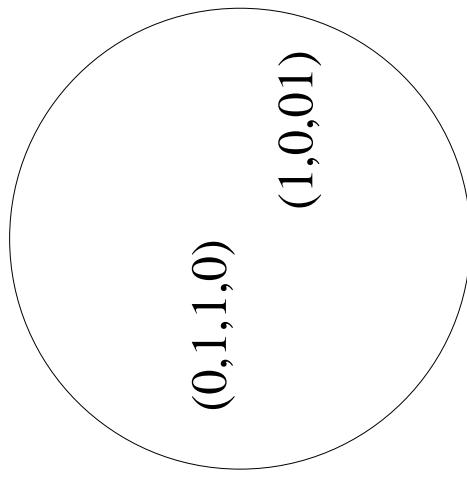


## Hiding Internal Wires

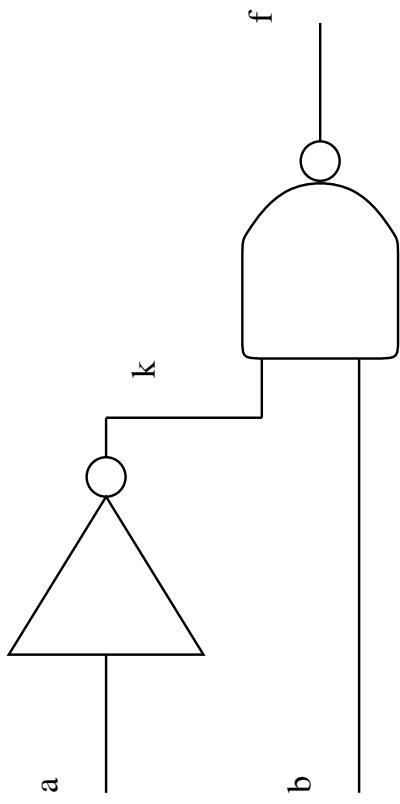


$$\text{circuit1}(\text{in}, \text{out}) = \exists k : \text{INV}(\text{in}, k) \wedge \text{INV}(k, \text{out})$$

That definition also defines the set:



## Specification Style - Assertion



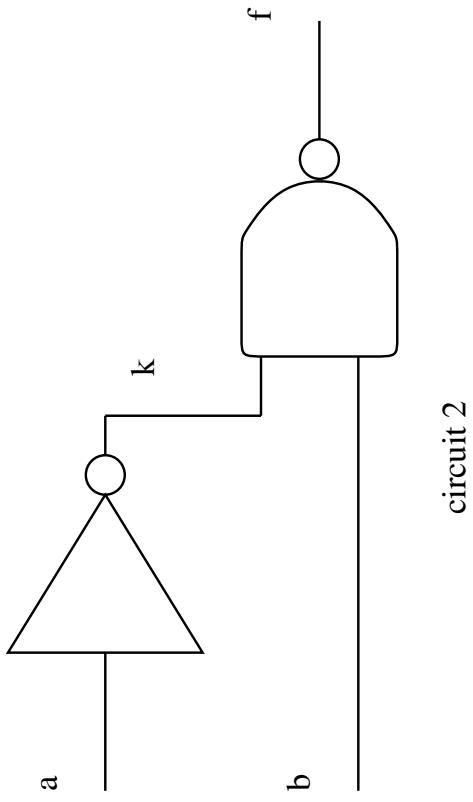
circuit 2

$\text{INV}(\text{in}, \text{out}) = (\text{out} = \neg \text{in})$

$\text{NAND}(\text{in1}, \text{in2}, \text{out}) = (\text{out} = \neg(\text{in1} \wedge \text{in2}))$

$\text{circuit2}(a, b, f) = \exists k : \text{INV}(a, k) \wedge \text{NAND}(k, b, f)$

## Specification Style - Functional



$\text{INV}(\text{in}) = \neg \text{in}$

$\text{NAND}(\text{in1}, \text{in2}) = \neg(\text{in1} \wedge \text{in2})$

$\text{circuit2(a,b)} = \text{NAND}(\text{INV}(\text{a}), \text{b})$

$\text{circuit2(a,b)} = \text{let } k = \text{INV}(\text{a}) \text{ in } \text{NAND}(k, \text{b})$

## Homework

Get account in Unix machines or install your own version of PVS.

Work some proves.